



Analizador Léxico, Sintáctico,
Semántico y Código Intermedio =
Compilador.

Macrogrupo 1.

Grupo 1
Grupo 3
Grupo 4
Grupo 11



Analizador Léxico.

- 4 scanners:
 - 2 implementados a mano (grupos 4 y 11)
 - 1 implementado con Coco/R (grupo 3)
 - 1 implementado con JFlex (grupo 1)
- Se elige el scanner implementado con JFlex (Grupo 1).
 - Comodidad a la hora de cambiar código.
 - Fácil integración con el SLK.
 - Experiencias anteriores.



Analizador Léxico.

Tipos de tokens generados:

Palabras Reservadas

Atributo: Puntero a la Tabla de Símbolos

Ejemplos: FOR, IF, WITH

Identificadores

Atributo: Puntero a la Tabla de Símbolos

Ejemplos: variable, nota, casa

Operadores comparación

Atributo: Tipo Enumerado

Ejemplos: <, <=, >, >=, !=, =

Operador Asignación

Atributo: Nada

Ejemplos: :=

Operador Unitario

Atributo: Enumerado



Analizador Léxico.

Tipos de tokens generados:

- Operadores Aditivos

 - Atributo: Tipo Enumerado

 - Ejemplos: +, -

- Operadores Multiplicativos

 - Atributo: Tipo Enumerado

 - Ejemplos: *, /

- Número Entero (octal y hexadecimal)

 - Atributo: Valor numérico

 - Ejemplos: 1, 56

- Número Real

 - Atributo: Valor numérico

 - Ejemplos: 1.2, 56.7893

- EOF



Analizador Léxico.

Tipos de tokens generados:

Cadenas de caracteres

Atributo: Puntero a la tabla de símbolos

Ejemplos: "Hola mundo", "adios"

Carácter (en octal)

Atributo: Carácter

Ejemplos: a, b, z, 3

Puntuación

Atributo: Tipo enumerado

Ejemplos: ;, (,]

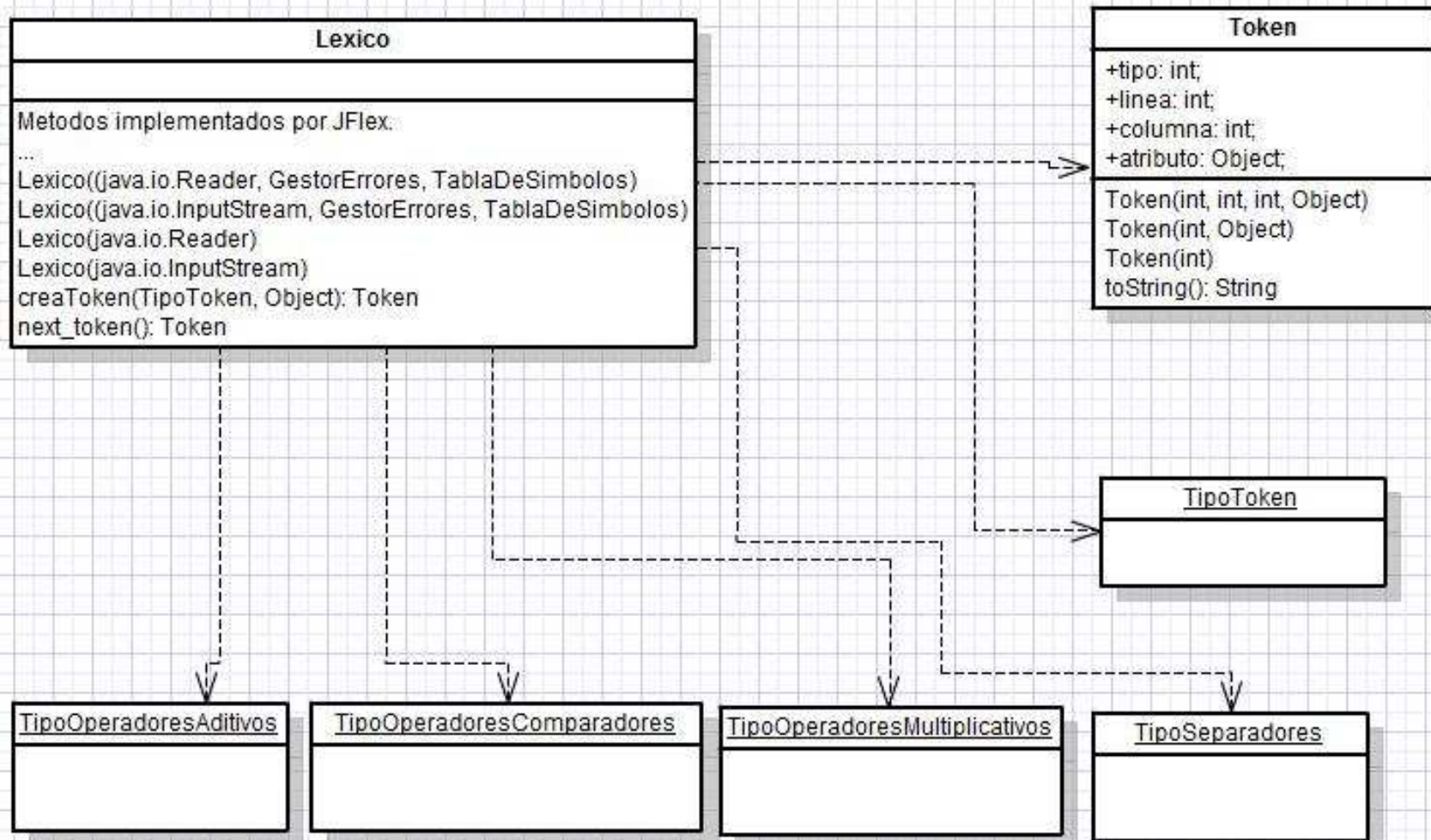
Función predefinida

Atributo: Tipo enumerado

Procedimiento predefinido

Atributo: Tipo enumerado

Analizador Léxico.





Analizador Léxico.

Herramienta utilizada: JFlex.

JFlex es un generador de scanners para Java, implementado también en Java, que genera un analizador léxico de manera rápida y sencilla, independientemente de la plataforma que utilicemos a la hora de programar.

Recibe archivo .flex → Genera AFD.



Analizador Léxico.

Herramienta utilizada: JFlex.

Se utilizan:

- PATRONES
- ESTADOS
- FUNCIONES
- Código Java empotrado

Nuestro Archivo .flex



Diseño de la tabla de símbolos

Tabla hash por cada ámbito
con estructura de árbol

Diseño de la tabla de símbolos

```

MODULE mod0;
  VAR a : INTEGER;
  MODULE mod1
    VAR b,c: INTEGER
    EXPORT c;
    PROCEDURE incrementar (VAR &c : INTEGER);

      BEGIN

        b := 1 + c;
      END p;

    BEGIN
      incrementar(c);
    END mod1.
  BEGIN
    a := c;
  END mod0.
  
```

Lista de exportadas
null

mod0								
Lexema	Tipo	Valor	Contenido	contiene	Nº args	Tipo args	Paso args	Mod padre
a	INTEGER		null	null	0	Null	Null	Null

Diseño de la tabla de símbolos

```

MODULE mod0;
  VAR a : INTEGER;
  MODULE mod1
    VAR b,c: INTEGER
    EXPORT c;
    PROCEDURE incrementar (VAR &c : INTEGER);

      BEGIN

        b := 1 + c;
      END p;

    BEGIN
      incrementar(c);
    END mod1.
  BEGIN
    a := c;
  END mod0.
  
```

Lista de exportadas
null

mod0								
Lexema	Tipo	Valor	Contenido	conteniente	Nº args	Tipo args	Paso args	Mod padre
a	INTEGER		null	null	0	Null	Null	Null
mod1	MODULE			Null	0	Null	Null	Null

Diseño de la tabla de símbolos

```
MODULE mod0;  
  VAR a : INTEGER;  
  MODULE mod1  
    VAR b,c: INTEGER  
    EXPORT c;  
    PROCEDURE incrementar (VAR &c : INTEGER);  
  
      BEGIN  
  
        b := 1 + c;  
      END p;  
  
    BEGIN  
      incrementar(c);  
    END mod1.  
  BEGIN  
    a := c;  
  END mod0.
```

Puntero a la
tabla de
simbolos de
mod1

Lista de exportadas
null

mod0								
Lexema	Tipo	Valor	Contenido	conteniente	Nº args	Tipo args	Paso args	Mod padre
a	INTEGER	Null	Null	null	0	Null	Null	Null
mod1	MODULE		Null	Null	0	Null	Null	Null

Diseño de la tabla de símbolos

```
MODULE mod0;  
  VAR a : INTEGER;  
  MODULE mod1  
    VAR b,c: INTEGER  
    EXPORT c;  
    PROCEDURE incrementar (VAR &c : INTEGER);  
      BEGIN  
        b := 1 + c;  
      END p;  
      BEGIN  
        incrementar(c);  
      END mod1.  
  END mod1.  
BEGIN  
  a := c;  
END mod0.
```

Puntero a la
tabla de
símbolos de
mod0

Puntero a la
tabla de
símbolos de
incrementar

Lista de exportadas
{ c }

mod1								
Lexema	Tipo	Valor	Contenido	conteniente	Nº args	Tipo args	Paso args	Mod padre
b	INTEGER	Null	Null	Null	0	Null	Null	
c	INTEGER	Null	Null	Null	0	Null	Null	
Incrementar	PROCEDURE	Null		Null	1	{INTEGER}	{VALOR}	

Diseño de la tabla de símbolos

```
MODULE mod0;  
  VAR a : INTEGER;  
  MODULE mod1  
    VAR b,c: INTEGER  
    EXPORT c;  
    PROCEDURE incrementar (VAR &c : INTEGER);
```

BEGIN

b := 1 + c;

END p;

BEGIN

incrementar(c);

END mod1.

BEGIN

a := c;

END mod0.

Puntero a la
tabla de
símbolos de
mod1

Lista de exportadas
null

En esta linea el compilador pondria como ambito actual el del modulo 1

En esta linea el compilador buscaria la variable b en la tabla del modulo 1 para guardar el valor de 1 + c.

incrementar								
Lexema	Tipo	Valor	Contenido	contiene	Nº args	Tipo args	Paso args	Mod padre
c	INTEGER	Null	Null		0	Null	Null	Null

Diseño de la tabla de símbolos

```
MODULE mod0;  
  VAR a : INTEGER;  
  MODULE mod1  
    VAR b,c: INTEGER  
    EXPORT c;  
    PROCEDURE incrementar (VAR &c : INTEGER);  
  
      BEGIN  
  
        b := 1 + c;  
      END p;  
  
    BEGIN  
      incrementar(c);  
    END mod1.  
  BEGIN  
    a := c;  
  END mod0.
```

Puntero a la
tabla de
símbolos de
mod0

Puntero a la
tabla de
símbolos de
incrementar

En esta línea el compilador pondría como
ámbito actual el del mod0

Lista de exportadas
{ c }

mod1								
Lexema	Tipo	Valor	Contenido	conteniente	Nº args	Tipo args	Paso args	Mod padre
b	INTEGER	Null	Null	Null	0	Null	Null	
c	INTEGER	Incrementar(c)	Null	Null	0	Null	Null	
Incrementar	PROCEDURE	Null		Null	1	{INTEGER}	{VALOR}	

Diseño de la tabla de símbolos

```

MODULE mod0;
  VAR a : INTEGER;
  MODULE mod1
    VAR b,c: INTEGER
    EXPORT c;
    PROCEDURE incrementar (VAR &c : INTEGER);

      BEGIN
        b := 1 + c;
      END p;

    BEGIN
      incrementar(c);
    END mod1.
  BEGIN
    a := c;
  END mod0.
  
```

Exportada del modulo 1

Puntero a la
tabla de
símbolos de
mod1


Lista de exportadas
null

mod0								
Lexema	Tipo	Valor	Contenido	conteniente	Nº args	Tipo args	Paso args	Mod padre
a	INTEGER	Valor de c	null	null	0	Null	Null	Null
mod1	MODULE		Null	Null	0	Null	Null	Null
c	INTEGER	c(exportado de mod1)	Null	Null	0	Null	Null	Null




Interfaz de acceso de la T.S. (analizador léxico)

- Inserta_Clave(String, Tipo-Token)
 - Token=(Tipo-Token, Atrib)
 - El analizador léxico llamará a esta función para insertar nuevos identificadores
- Busca_Clave(String)
 - Busca la cadena que se le pasa como parámetro en la tabla de símbolos



Interfaz de acceso de la T.S. (analizador sintáctico y semántico)

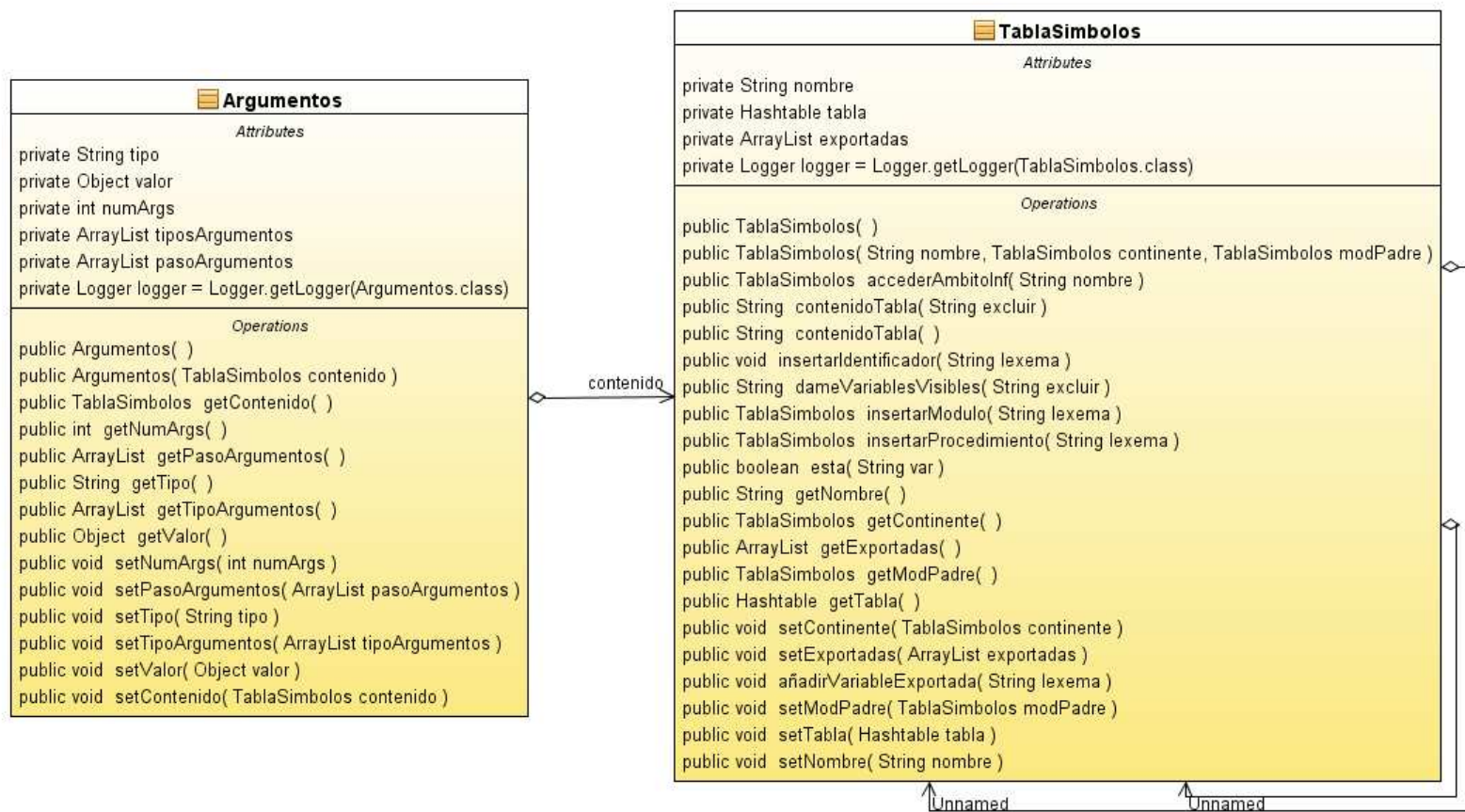
- \wedge TS CreaTabla(\wedge TS) :
 - Devuelve un puntero a la nueva tabla creada, y se le pasa como parámetro el puntero de la tabla padre
- \wedge TS CierraTabla(\wedge TS):
 - Devuelve un puntero a la nueva tabla activa, que será la que era continente de la del parámetro \wedge TS



Interfaz de acceso de la T.S. (analizador sintáctico y semántico)

- ^TS Completa (Campo, Valor, String) :
 - Rellena campos de la tabla de símbolos con el valor pasado como parámetro.
- Valor Consulta (Campo, String):
 - Devuelve el valor del identificador pasado como parámetro para el tipo de campo "Campo".

Diagrama de clases





Analizador sintáctico.

- **Especificación y diseño del Parser.**

Analizador descendente predictivo tabular.

Herramienta de generación Slk.

Sintaxis particular de slk muy próxima a la notación EBNF.

Se obtiene un código que implementa el Parser.



Analizador sintáctico.

- **Gramática.**

Factorizada y sin recursión por la izquierda, LL(1).

Sigue unas reglas de notación que ayudan al mejor entendimiento y legibilidad de la misma:

- No terminales siguen el estilo de los identificadores de Java.

Ejemplo: `noTerminal`, `otroNoTerminal`.

- Los identificadores de los símbolos terminales aparecen con todas sus letras en mayúsculas

Ejemplo: `TERMINAL`, `OTROTERMINAL`

- Cada producción debe ir en una nueva línea. Las producciones con distinta parte izquierda deben estar separadas por al menos un salto de línea.

- El símbolo vacío se representa por `_epsilon_`



Analizador sintáctico.

- **Herramientas consideradas y elección**

Un analizador sintáctico descendente predictivo tabular es más eficiente debido a que no se apoya en la recursión.

Se ha elegido la herramienta SLK ya que simplifica diseño e implementación al comprobar que la gramática es LL(1) y al generar la tabla de análisis.

Inconvenientes:

- no tener control sobre el código generado.

SLK toma como entrada una gramática LL(1) en formato EBNF. Genera clases con código propio y otras que el usuario debe desarrollar, como `SlkAction` o `SlkLog`.



Analizador sintáctico.

- **Clases generadas por SLK.**
 - **SlkParser:** Clase principal del analizador.
 - **SlkConstants:** traducción de los símbolos de la gramática.
 - **SlkToken:** Clase proporcionada por el usuario, nexo de unión entre el analizador sintáctico y el analizador léxico.
 - **SlkError:** un método por cada tipo de error sintáctico: Mismatch, No Entry, Input left.
 - **SlkString:** transformar los códigos que utiliza el parser en sus descripciones.
 - **SlkAction:** acciones semánticas.
 - **SlkLog:** trace, trace_production, trace_action.



Diseño del Gestor de Errores

- Construcción del Gestor de Errores
- Diseño e Interfaces del Gestor de Errores
- Diagrama de clases



Construcción del Gestor de Errores

- Se pueden encontrar errores en cada fase de las que consta el compilador.
 - Detectar el primer error producido y pararse.
 - Poder continuar la compilación y permitir la detección de más errores en el programa fuente.



Construcción del Gestor de Errores

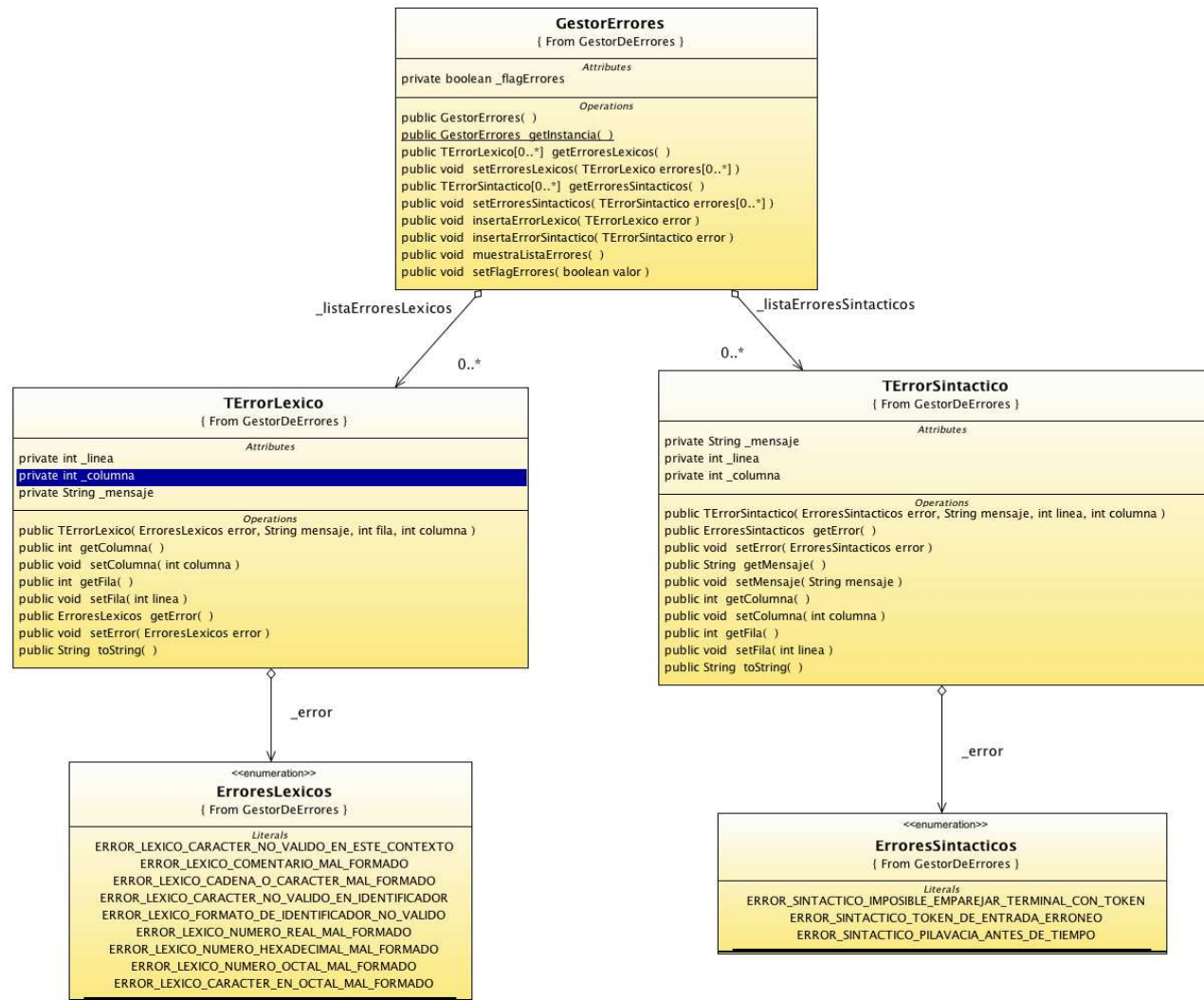
- La fase de análisis léxico detecta errores producidos por los caracteres restantes de la entrada que no forman ningún componente léxico del lenguaje.
- Los errores donde la cadena de componenetes léxicos violan las reglas de estructura (sintaxis) del lenguaje son determinados por la fase del análisis sintáctico.
- Durante el análisis semántico el compilador intenta detectar construcciones que aunque tengan la estructura sintáctica correcta, no tengan significado para la operación implicada.



Diseño e interfaces del Gestor de Errores

- El diseño del Gestor de Errores se hace pensando en la futura integración con el resto de los módulos del compilador.
 - Lograr la máxima cohesión y el mínimo acoplamiento posible.
 - Reducir la complejidad.
 - Mayor flexibilidad frente a cambios o modificaciones.
- Encapsulamiento del tipo de error mostrado al usuario.
- Los módulos que utilicen el Gestor de errores sólo deberán preocuparse por utilizar el tipo de manejador adecuado.

Diagrama de clases





Analizador semántico.

- **Especificación y diseño del analizador semántico.**
 - Fase donde relacionamos la información que calculamos con el significado del programa.
 - Determinar la información que no puede ser descrita por las GIC.
 - Asociamos la información a las construcciones del lenguaje de programación proporcionando atributos a los símbolos de la gramática.
 - Los valores de los atributos se calculan mediante reglas semánticas asociadas a las producciones gramaticales.
 - Completar la información de los símbolos que tenemos en la TS.



Analizador semántico.

- Se distinguen dos tipos de análisis semántico:
 - Estático
 - Comprobación de tipos
 - Comprobaciones asociadas al flujo del control
 - Comprobaciones de unicidad
 - Comprobaciones relativas a nombres
 - Dinámico
- Recorreremos en un determinado orden y calcularemos en cada nodo la información semántica necesaria. (1 sola pasada)
- Evaluación de los atributos mediante métodos basados en reglas.
- Como resultado: árbol sintáctico anotado.



Analizador semántico.

- **Herramientas utilizadas**

- No ha sido necesaria considerar otras.
- SLK.

Se ha elegido la herramienta SLK ya que fue la utilizada para el análisis sintáctico, de modo que la salida de esta fase se convierte en la entrada de la fase del análisis semántico.

Inconvenientes:

- No tener control sobre el código generado.

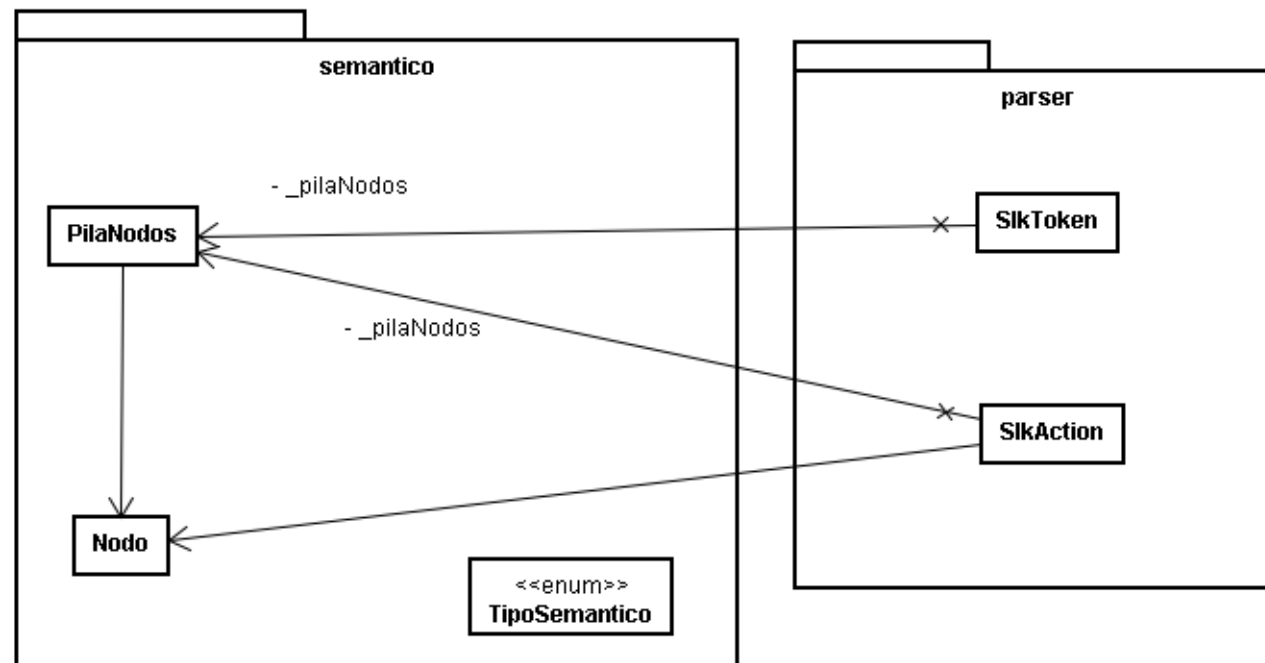
SLK toma como entrada una gramática LL(1) en formato EBNF. Las clases que debemos implementar para esta fase es la SLKAction, además de otras propias.



Analizador semántico.

- **Clases generadas por SLK.**
 - **SlkAction:** contiene todos los métodos que determinan la acción a realizar en cada caso según su análisis semántico (80).
 - **Nodo:** estructura que contiene toda la información recogida en cada acción semántica.
 - **PilaNodos:** pila para almacenar los nodos con la información que se completa en cada acción semántica.
 - **TipoSemantico:** clase enumerado que contiene todos los tipos semánticos considerados.
- Así, como la utilización de la propia TS y el GE.

Analizador semántico.



- Relaciones mas importantes entre el semántico y el sintáctico. Se omiten las relaciones con la TS y el GE.



Generación de código intermedio.

- **Especificación y diseño**

- El generador de código ensamblador se ha implementado como complemento al resto de actividades sintácticas dentro de las acciones semánticas ofrecidas por la herramienta SLK.
- La generación es empotrada, por tanto, con la diferenciación de que ha sido implementada una nueva clase, Generador, que será la que genere el código propiamente dicho y presente una interfaz amigable dentro de las acciones definidas previamente.
- La generación de código se produce en las acciones semánticas que proceda, obteniendo un fichero de salida con los resultados.



Generación de código intermedio.

- **Herramientas utilizadas**

- Uno de los propuestos en clase
 - ENS2001.
- Integra la función de **Ensamblador**, de un subconjunto de instrucciones del estándar IEEE 694.
- Tiene un **Simulador**, ya que es capaz de ejecutar programas ensamblados para dicha implementación particular del estándar.

Generación de código intermedio.

Generador

Atributos

```
public Ventana _interfaz
private String _codigo
private PrintStream _file
private FileOutputStream _codObj
private int _contadorEtiquetas
private Integer _contadorVariables[0..*]
private String _pilaListaVariables[0..*,0..*]
private TablaDeSimbolos _tabla
private boolean _etiquetaUltimaEmision
private int _SEPCODIGO = 25
private long _contadorTemporales = 0
```

Operations

```
public Generador( TablaDeSimbolos ts, String fout )
public void emite( String arg0 )
public void emiteEtq( String arg0 )
public void emite( String arg0, String com )
public void anadeAlComienzo( String arg0 )
public void escribeSeccion( )
public String dameNuevaEtiqueta( )
public String dameNuevaTemp( String nombre, int size )
public void abreAmbito( )
public void cierraAmbito( )
public int getTamanoTotalVariables( boolean esFuncion )
public String generaCodigoAritmetica( Nodo sumando1, Nodo sumando2, Nodo operador )
public void generaCodigoMultiplicaciones( Nodo factores[0..*] )
public void generaCodigoAsignacion( Nodo destino, Nodo origen )
public void generaCodigoComparacion( Nodo comparables[0..*], Nodo operaciones[0..*], Nodo resultado )
public void generaCodigoBooleano( Nodo resultado )
public void generaCodigoLlamadaAFuncion( String nombre, Nodo resultado )
public void generaCodigoSubprograma( String nombre, boolean esFuncion )
public void generaCodigoEntrada( Nodo identificador )
public void generaCodigoSalida( Nodo identificador )
private String generaNuevoLiteral( TipoSemantico tipoSemantico[0..*], String lexema )
private int getPosicionReal( Nodo arg0 )
```